Name:

ID:

CSCI 3104, Algorithms **Profs. Chen & Grochow**

Problem Set 9 – Due Fri Apr 10 11:55pm **Spring 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- All submissions must be typed.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please allot at least as many pages per problem (or subproblem) as are allotted in this template.

Quicklinks: 1a 1b 1c 2a 2b

Name:

ID:

CSCI 3104, Algorithms **Profs. Chen & Grochow**

Problem Set 9 – Due Fri Apr 10 11:55pm **Spring 2020, CU-Boulder**

1. Could dynamic programming be applied to give an efficient solution to the following problems? Justify your answer in terms of the optimal substructure property on the overlapping sub-problems. Even if you think the answer is yes, you do not need to give an algorithm; the question is *not* "show us how to solve it using DP", it is "give an argument as to whether DP is a reasonable approach to try here."

   (a) List maximum.
   Input: List L of numbers.
   Output: The maximum element in the list.

   Solution We can define a recursive solution to the find-max problem as:

$$M(i) = \max \begin{cases} M(i-1) \\ L[i] \end{cases}$$

   This can be implemented as a linear time, bottom-up algorithm, i.e. starting with $i = 1$ and iterating up to $n = |L|$. Observe that this bottom-up DP algorithm is precisely a linear scan of the array. Thus, DP is a reasonable and efficient approach in terms of time complexity. By storing only the current maximum element, we have $\mathcal{O}(1)$ space complexity.

(b) Rod cutting.

Input: A list of values $v_1, \ldots, v_n$ for rods of length $1, \ldots, n$, respectively.

Goal: Divide a rod of length n into pieces of lengths $\ell_1, \ldots, \ell_k$ (k can vary) to maximize the total value $\sum_{i=1}^{k} v_{\ell_i}$.

Note: While this problem is discussed on GeeksForGeeks (and elsewhere), the explanation of optimal substructure on GeeksForGeeks, while not incorrect, is not sufficient explanation to demonstrate mastery of this question.

Solution Let $r_n = \sum_{i=1}^{k} v_{\ell_i}$ for k cuts of a rod of length n.
Then we define the recurrence for the optimal solution as:

$$r_n^* = \max_{1 \leq i \leq n} \{v_i + r_{n-i}\}$$

To solve this problem, we will do it in 2 steps: first, cut 1 piece out and sell that; then cut the remaining piece to get the max value of that one. In the first step, we can cut at length $1, ...n$ and the corresponding value is $v_1, ...v_n$, and the remaining part will have length $n-1, n-2, ..., 0$.

(c) Graph 3-coloring.

Input: A simple, undirected graph G.

Goal: Decide whether one can assign the colors $\{R, G, B\}$ to the vertices of G in such a way that no two neighbors get the same color.

Hint: You may find the notion of critical graph useful.

Solution: For full credit, students needed to point to a specific type of subproblem and argue as to whether that type of subproblem was useful in designing a DP algorithm.

- **Solution 1:** The most natural type of sub-problem is to consider arbitrary sub-graphs. We note that it is difficult to extend an optimal coloring of a subgraph H to an optimal coloring of the parent graph G, as there are additional edges between the vertices of H and the vertices not in H.

  Remark: Critical graphs are nice, as we can simply consider arbitrary subgraphs as our type of sub-problem when designing a DP algorithm for coloring.

- **Solution 2:** We may design a dynamic programming algorithm by leveraging independent sets. We note that a coloring φ of the vertices of the graph G partitions $V(G)$ into color classes. Observe that no two vertices receiving the same color under φ are adjacent. So the color classes form independent sets.

  The idea for the algorithm is to select a maximal independent set I, which is our first color class, then color the graph $G[V - I]$ induced by the vertices of $V - I$. We examine over all such maximal independent sets (MIS) to ensure an optimal for $G[V - I]$. Denote $\chi(G)$ as the chromatic number of G, or the minimum number of colors required to properly color $V(G)$. The recurrence of the optimal solution is

$$\chi(G) = \begin{cases} 1 & : G \text{ is an independent set} \\ 1 + \min\limits_{I \text{ MIS of } G} \chi(G[V - I]) & : \text{otherwise.} \end{cases}$$

  Remark: We remark that computing $\chi(G)$ using this recurrence takes exponential time. However, deciding if G is k-colorable (for $k \geq 3$) is a provably difficult problem. Precisely, the k-colorability decision problem is NP-Complete. We do not expect a polynomial time algorithm to exist.

Name:

ID:

CSCI 3104, Algorithms **Profs. Chen & Grochow**

Problem Set 9 – Due Fri Apr 10 11:55pm **Spring 2020, CU-Boulder**

2. Write down the recurrence for the optimal solution for each of the following problems. Justify your answer.

   (a) Social distancing gold-panning. Imagine a river network in which your team can pan for gold, but no two of you can stand in adjacent positions. You have some idea of the expected amount $w(v)$ of gold you will find at each location v, but must decide in which locations your team should look.

   Input: A rooted tree T, with root vertex $r \in V(T)$, and vertex weights $w\colon V(T) \to \mathbb{R}_{\geq 0}$

   Output: A subset of vertices $P \subseteq V(T)$ such that no two vertices in P are adjacent, and maximizing the value $\sum_{v \in P} w(v)$.

   Solution We define the recurrence $\mathrm{Opt}(v)$ where $v(T)$ starting with $v = r$.

$$\mathrm{Opt}(v) = \begin{cases} w(v) & \text{v is a leaf} \\ \max \begin{cases} w(v) + \sum_{u \in G(v)} \mathrm{Opt}(u) \\ \sum_{u \in C(v)} \mathrm{Opt}(u) \end{cases} & \text{otherwise} \end{cases}$$

   where $C(v)$ denotes the children of v and $G(v)$ denotes the grandchildren of v.

(b) Counting Knapsack. Here, we are considering the knapsack problem, but rather than returning the value of the optimum knapsack, or the optimum knapsack set, we are asking for the *number* of different optimum knapsacks (which all therefore have the same value).

Input: A list $L = [(w_1, v_1), \ldots, (w_n, v_n)]$, and a threshold weight W.

Output: The count of max-value knapsacks. A max-value knapsack is a subset $S \subseteq \{1, \ldots, n\}$ such that (1) $\sum_{i \in S} w_i \leq W$, and (2) the value $\sum_{i \in S} v_i$ is maximum among all subsets satisfying (1). The output should be *how many* different optimal solutions S there are.

For instance, if $L = [(1,1), (1,2), (2,2), (2,3)]$ and $W = 2$, then the output would be 2, because there are two optimal solutions: taking either the first two items or the very last item results in a valid knapsack of value 3. (Note that this is *not* just the total number of valid knapsacks; in this case there is a third set that fits within the weight threshold, namely the singleton $\{(2,2)\}$, but that set does not have optimal value.)

Remark: This problem was modified after the fact. We accepted solutions for either of the following modified problems.

(a) Count the number of ways to fill a knapsack with weight exactly W using the n input items.

(b) Describe how to use the lookup table for the Knapsack problem from class to count the number of Knapsack configurations achieving the maximum value.

Solution (a): Our goal is to count the number of ways to fill a knapsack with weight exactly W using items $1, \ldots, k$ where $1 \leq k \leq n$. We set up the recurrence in terms of W and item k. For the recursive step, we have two cases: selecting k or not selecting k.

- **Case 1:** If we choose not to include item k in our knapsack, then our items are drawn from the first $k-1$ items. There are $F(W, k-1)$ ways to to fill the knapsack with weight W using items $1, \ldots, k-1$.
- **Case 2:** Suppose instead we include item k in our knapsack. Then we need to select from items $1, \ldots, k-1$ to fill the knapsack to the remaining capacity of $W - w(k)$. There are $F(W - w(k), k-1)$ selections.

The set of configurations from Case 1 and the set of configurations from Case 2 share no common configurations. So by the rule of sum, we add to obtain:

$$F(W, k) = F(W, k-1) + F(W - w(k), k-1).$$

We now turn our attention to the base cases, which are included in the recurrence below.

$$F(W,k) = \begin{cases} 0 & n < 0 \vee W < 0 \\ 1 & W = 0 \wedge n = 0 \\ F(W, k-1) + F(W - w(k), k-1) & \text{otherwise} \end{cases}$$